Enterprise Architektur-Muster

JULIAN BRUDER*, ABDELLAH FILALI*, and LUCA FRANKE*, Hochschule für Technik, Wirtschaft und Kultur Leipzig (HTWK Leipzig), Deutschland

In diesem Papier werden verschiedene Enterprise Architektur-Muster und deren Rolle in modernen Geschäftsprozessen untersucht und anschließend unter Einbeziehung technischer und struktureller Eigenschaften anhand ihrer Agilität bewertet. Dabei orientiert sich die Reihenfolge der Betrachtung jener Architektur-Muster am historischen Verlauf derer Entwicklung und der Notwendigkeit dieser. Genauer werden die monolithische Architektur, modulare monolithische Architektur, service-orientierte Architektur, Microservice-Architektur, Layered-Architecture, Event-Driven Architecture, cloud-native Architektur und die Microkernel-Architektur betrachtet.

Insgesamt zeigt sich, dass klassische Enterprise Architektur-Muster zwar mit geringer initialer Komplexität punkten, mit weiterführender Entwicklung allerdings Flexibilitätsprobleme bedingen. Dem entgegen zeichnen sich die modernen Architektur-Muster durch hohe Agilität und damit hoher Flexibilität gegenüber den in der modernen Geschäftswelt ständig wechselnden Anforderungen aus. Besonders die cloud-native Architektur wird diesen Anforderungen gerecht.

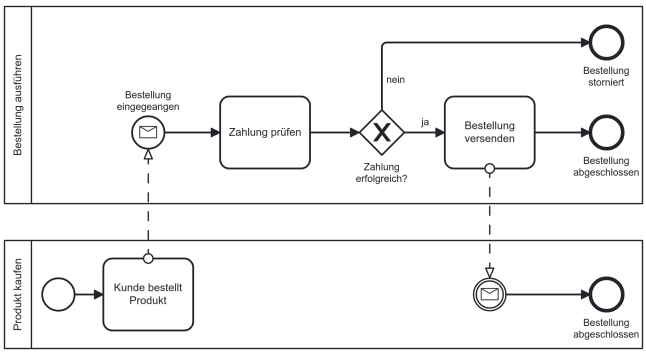

Abb. 1. Geschäftsprozessmodell des MVPs des E-Commerce-Beispiels

1 EINLEITUNG

Moderne Software-Systeme bestehen aus verschiedenen Komponenten [Salah et al. 2016]. Dabei nimmt der Nutzer des Systems jenes zwar als eine Software wahr, jedoch verbirgt sich in Realität meist eine Struktur aus Software-Komponenten und deren Beziehungen hinter dieser Wahrnehmung - ein verteiltes Softwaresystem. Die Art und Weise dieser Struktur und Beziehungen wird als Architektur bezeichnet. Wie wird eine solche Architektur mit all ihren Anforderungen der modernen und schnelllebigen Geschäftswelt konstruiert?

Betrachten wir folgend ein Anwendungsbeispiel zur Motivation. In einem Tech-Startup soll ein Backend für ein internationales E-Commerce-System entwickelt werden. Dabei werden folgende Anforderungen gestellt:

- Zukünftig viele Nutzer und hoher Traffic erwartet
- Geringes Kapital für Infrastruktur
- Rechtliche Regularien sind teilweise unklar
- Hohe Sicherheitsanforderungen aufgrund des Cash-Flows
- Agiles Team von acht fähigen Entwicklern
- Geldgeber wollen erste Auslieferung in zwei Wochen

Das Team arbeitet agil nach Scrum und einigt sich in einer der ersten Planungsphasen auf ein *Minimum Viable Product* (kurz MVP) mit Bestellung, Bezahlung und Versand. Abbildung 1 zeigt die Modellierung des Geschäftsprozesses für diesen MVP. Die Anforderungen sind somit zwar grob strukturiert aber trotzdem ungewiss. Die rechtlichen Regularien sind unklar, was zu späten technischen Änderungen führen kann. Anfangs ist der erwartete Traffic vermutlich niedrig, später sollte dieser aus geschäftlicher Sicht bestenfalls ansteigen. Wie wird der erwartet hohe Traffic aus infrastruktureller Sicht vorgesehen und finanziert? Sind acht Entwickler ausreichend? Kann nach zwei Wochen schon geliefert werden?

All diese Fragen betreffen nicht nur die Software, sondern offensichtlich auch das Geschäft. Die Antworten dafür liefern Enterprise-Architekturen. Deren Ziel ist die Ausrichtung von Geschäft und IT, also die Unterstützung des Geschäfts durch die IT und umgekehrt [Greefhorst and Proper 2011]. Ein Enterprise Architektur-Muster ist dabei eine spezifische Strategie zur Umsetzung dieser Ausrichtung. Erwähnenswert ist hier die Nähe zur Bedeutung des lateinischen Wortes *architectura*. Wörtlich übersetzt ist es die *Wissenschaft der Baukunst* - meint aber sowohl das Produkt des Bauens als auch den Prozess des Bauens [1]. In Bezug dazu kann die Enterprise-Architektur also als Vision einer grundlegenden Struktur und Beziehungen von Komponenten eines Systems betrachtet werden. Wichtig ist hierbei die Abgrenzung zur Software-Architektur, der grundlegenden Struktur und Beziehungen von Teilen einer Software.

Im Verlauf der Arbeit werden verschiedene Enterprise Architektur-Muster und deren Rolle in modernen Geschäftsprozessen untersucht und anschließend unter Einbeziehung technischer und struktureller Eigenschaften anhand ihrer Agilität bewertet. Dabei wird auch immer auf das eingangs erwähnte und in Abbildung 1 dargestellte E-Commerce-Beispiel zurückgegriffen. Die Arbeit ist wie folgt strukturiert:

Zunächst wird in Abschnitt TODO auf die monolithische Architektur eingegangen, bevor die Modularisierung des Monolithen in Abschnitt TODO diskutiert wird. Danach wird in Abschnitt TODO die service-orientierte Architektur betrachtet. Folgend wird in Abschnitt TODO die Microservice-Architektur diskutiert. Anschließend wird in Abschnitt TODO untersucht, wie eine Schichten-Architektur andere, dienst-basierte Architekturen ergänzen kann. Daraufhin werden sowohl die event-basierte Architektur in Abschnitt TODO als auch die cloud-native Architektur in Abschnitt TODO erläutert. Als letztes Architektur-Muster wird in Abschnitt

*Alle Studierenden trugen zu gleichen Teilen zu dieser Arbeit bei.

Diese Arbeit wurde im Rahmen des Mastermoduls „Software Engineering“ (Dozent: Prof. Dr. Andreas Both) an der HTWK Leipzig im Wintersemester 2024/2025 erstellt.

[1] https://www.duden.de/rechtschreibung/Architektur, abgerufen am 17.01.2025

TODO die Microkernel-Architektur betrachtet. Abschließend werden die Architekturen in Abschnitt TODO nach technischen und agilen Kriterien verglichen und folglich die Ergebnisse dieser Arbeit zusammengefasst.

2 KLASSISCHE ENTERPRISE-ARCHITEKTUREN

2.1 Monolithic Architecture

Der Begriff *Monolith*, abgeleitet aus dem Altgriechischen und mit der Bedeutung *einheitlicher Stein*[2], wird in der Softwarearchitektur verwendet, um ein Designmuster zu beschreiben, bei dem die gesamte Funktionalität einer Anwendung in einem einzigen, zusammenhängenden System integriert ist. In dieser Architektur übernimmt ein einzelner Prozess die Ausführung der gesamten Anwendung [Chen et al. 2017, 1].

Anwendungen, die diesem Architekturansatz folgen, bestehen aus eng miteinander verknüpften Komponenten, die wechselseitig voneinander abhängen. Diese Komponenten können weder unabhängig betrieben, noch in bestimmten Fällen, isoliert kompiliert werden [Salaheddin and Ahmed 2022, 485].

Diese Architektur bietet insbesondere für kleinere Anwendungen mehrere Vorteile, wie etwa eine vereinfachte Testbarkeit, Logging, Deployment und Debugging. Ein weiterer Vorteil besteht darin, dass keine separate Datenbanksynchronisation erforderlich ist, da sämtliche Daten in einer einzigen Datenbank gespeichert werden [Blinowski et al. 2022, 20358].

Betrachten wir das E-Commerce-Beispiel . Dafür definieren wir drei Klassen (Siehe Abb. 2):

- `OrderService`: Klasse, die die Bestellungen verwaltet
- `PaymentService`: Klasse, die die Zahlungen abwickelt
- `ShipmentService`: Klasse, die die Lieferungen initiiert

Abb. 2. Monolith Architektur

Die Kommunikation zwischen den Klassen erfolgt über Methodenaufrufe. In diesem Zusammenhang stellt die Klasse `OrderService` die zentrale Komponente dar, die die Methoden anderer Klassen nutzt, um den Bestellprozess zu steuern. Ein wesentlicher Vorteil dieses Ansatzes liegt in der einfachen Kommunikation zwischen den Komponenten. Durch den direkten Einsatz von Methodenaufrufen wird die Komplexität verringert, die typischerweise mit der Interaktion zwischen verschiedenen verteilte Systemen verbunden wäre.

Eine beispielhafte Implementierung des `PaymentServices` in einer monolithischen Architektur ist im Anhang A.1 dargestellt. Der vollständige Quellcode des E-Commerce-Beispiels kann zudem bei GitHub[3] eingesehen werden.

[2]https://www.duden.de/rechtschreibung/Monolith, abgerufen am 17.01.2025
[3]https://github.com/Beleg-6-EAP/demo-monolith-ecommerce

Mit dem Wachstum und der zunehmenden Komplexität der Code-Basis treten jedoch einige Nachteile auf. Änderungen an einer Komponente können unerwartete kaskadierende Fehler auslösen, was die Weiterentwicklung in kleinen, autonomen Teams erschwert und verlangsamt.

Ein weiterer Nachteil ergibt sich aus der engen Kopplung zwischen den Komponenten, die durch direkte Methodenaufrufe entsteht. Änderungen an der Implementierung oder der Signatur einer Methode können in der Folge die gesamte Anwendung beeinflussen.

Zudem wird die Wiederverwendung von Funktionalitäten erschwert, da die Komponenten stark miteinander verknüpft sind. Dies führt zu einer Duplizierung von Code, was die Kosten für Änderungen erhöht, da diese an mehreren Stellen vorgenommen werden müssen.

Die daraus resultierende Komplexität und die erschwerte Wartung führen zu längeren Iterationen. Da das Deployment nur als Ganzes erfolgen kann und die Iterationen sich verlängern, kommt es zu seltenen Auslieferungen.

Ein weiteres Problem liegt in der erschwerten horizontalen Skalierung, da die Anwendung nur als Ganzes skaliert werden kann. Dies stellt eine Herausforderung dar, da bei wachsenden Anforderungen die gesamte Anwendung skaliert werden muss, anstatt einzelne Komponenten unabhängig voneinander zu skalieren.

Insgesamt ist die monolithische Architektur eine geeignete Lösung für kleinere Anwendungen, jedoch weniger geeignet für größere Systeme, da sie die Agilität und Flexibilität erheblich einschränken kann.

2.2 Modular Monolith

Das Hauptproblem der zuvor beschriebenen Architektur liegt in der engen Kopplung der Komponenten, was zu einer erhöhten Komplexität und begrenzten Flexibilität führt. Die Modular Monolithic Architecture stellt eine Weiterentwicklung der Monolithic Architecture dar, indem sie die Vorteile des monolithischen Ansatzes bewahrt und gleichzeitig die Nachteile der engen Kopplung verringert.

In diesem Architekturmodell wird die Code-Basis in mehrere Module unterteilt, die jeweils eine spezifische Teilfunktionalität der Anwendung implementieren.[Su and Li 2024, 11].

Die Modularisierung trägt zudem zur Reduzierung der Komplexität bei und ermöglicht eine bessere Organisation der Code-Basis, was sich positiv auf die Wartbarkeit der Anwendung auswirkt [Barde 2023, 23 - 24].

Betrachten wir erneut das E-Commerce-Beispiel. Diesmal wird die Anwendung in drei Hauptmodule aufgeteilt (Siehe Abb. 3):

- `order`: Modul, das für die Verwaltung der Bestellungen verantwortlich ist
- `payment`: Modul, das für die Abwicklung der Zahlungen verantwortlich ist
- `shipment`: Modul, das für die Initiierung der Lieferungen verantwortlich ist

Wie Abbildung 3 zeigt, ist jeder Service in einem eigenen Modul gekapselt. Ein konkretes Beispiel hierfür bietet die Implementierung des `OrderServices`, welche im Modul `order` realisiert ist (siehe Anhang A.2). Die Logik des `OrderService` ist vollständig innerhalb des entsprechenden Moduls integriert, während die Kommunikation

Abb. 3. Modular Monolith Architektur

mit anderen Modulen durch klar definierte Spezifikationen geregelt wird. Für weiterführende Informationen und die vollständige Implementierung des E-Commerce-Beispiels wird auf das GitHub-Repository verwiesen[4].

Durch die Modularisierung sind die Komponenten weniger eng miteinander gekoppelt, was eine verbesserte Zusammenarbeit in kleinen, autonomen Teams fördert, im Vergleich zu traditionellen monolithischen Ansätzen.

Ein Nachteil bleibt jedoch bestehen: Die Anwendung muss weiterhin als Ganzes deployed werden, was die Iterationen verlangsamt. Zudem bleibt die horizontale Skalierung weiterhin erschwert, und die Funktionalitäten können nicht wiederverwendet werden, da sie immer noch Teil einer einzigen Anwendung sind und eng miteinander gekoppelt bleiben.

3 MODERNE ENTERPRISE-ARCHITEKTUREN

3.1 Microkernel Architecture

Die Microkernel Architecture ist ein Architekturmuster, was sich durch Erweiterbarkeit, Flexibilität und vor allem Isolation der Funktionalitäten auszeichnet. Wie in Abbildung 4 dargestellt, enthält ein Microkernel zwei wesentliche Komponenten: Den Kern der Anwendung,der die wichtigsten grundlegenden Funktionalitäten bereitstellt und Module oder auch Plugins, die diesen Kern um Features erweitern [Richards 2015a, 21-22].

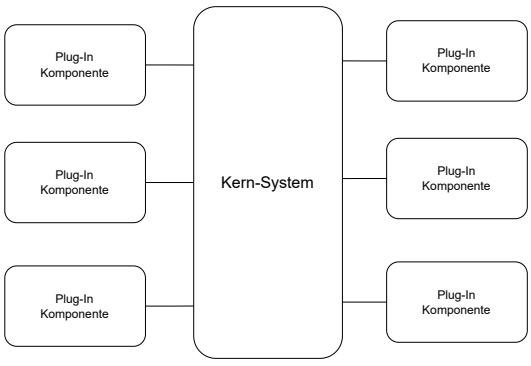

Abb. 4. Aufbau einer Microkernel Architecture

Der Kern der Anwendung implementiert dabei meist nur die minimalste Funktionalität, um die Anwendung oder das System lauffähig zu machen. Alle weiteren Funktionalitäten werden in Modulen implementiert, die auf den Kern aufbauen. Module sind meist unabhängig voneinander aufgebaut, es kann jedoch auch vorkommen, dass manche Module von anderen abhängig sind. Best practice für die Entwicklung von Microkernel Architekturen ist es, die Kommunikation zwischen einzelnen Modulen so gering wie möglich zu halten, um Probleme durch Abhängigkeiten zu vermeiden. Dadurch sind Module untereinander lose gekoppelt und können unabhängig voneinander entwickelt, getestet und deployed werden [Richards 2015a, 22].

Die Plugins können über verschiedene Wege mit dem Kern verbunden werden. Eine genaue Spezifikation zum Verbinden der Plugins mit dem Kern gibt es aber laut Architekturschema nicht, diese Entscheidung ist dem Entwickler überlassen und entsprechend der Anforderungen und Anwendungsumgebung zu treffen. Unabhängig von der Art der Verbindung definiert der Kern die Schnittstellen, um Plugins anzubinden. Diese Verbindung könnte dann beispielsweise über Web-Services, Messaging oder am einfachsten über direkte Objekt-Instanziierung innerhalb der gleichen Anwendung stattfinden [Richards 2015a, 22-23].

Zwischen Plugins und Kern werden Verträge definiert, die die Kommunikation zwischen den beiden Komponenten regeln. Diese Verträge können in Form von Interfaces, Klassen oder auch Datenstrukturen definiert werden. Alle Plugins, müssen sich zwingend an die definierten Verträge halten, um mit dem Kern kommunizieren zu können. Alternativ können auch Adapter verwendet werden, um bestehende Plugins an den Kern und die Verträge anzupassen, wodurch wiederrum die lose Kopplung der Komponenten verbessert wird.

Durch diesen Aufbau ergibt sich jedoch das Problem, dass der Kern jederzeit über Verfügbarkeit und Erreichbarkeit der Plugins informiert sein muss. Um dieses Problem zu lösen, kann eine zentrale Plugin-Registry verwendet werden. Diese Registry enthält alle aktuell verfügbaren Plugins sowie die dazugehörigen relevanten Informationen wie zum Beispiel Name des Service, Verträge, Verbindungsdetails, etc. Der Kern der Anwendung kann dann zur Laufzeit auf diese Registry zugreifen und Plugins dynamisch laden [Richards 2015a, 22].

Microkernel Architekturen können auch in andere Architekturmuster eingebettet werden, falls es nicht möglich sein sollte die gesamte Software in diesem Architekturmuster aufzubauen. Vor allem Teile von Anwendungen, die stark erweiterbar sein müssen, eignen sich gut für die Verwendung der Microkernel Architektur.

Ein klarer Vorteil dieses Architekturmusters ist die schnelle Reaktionsfähigkeit auf äußere Änderungen, da Anpassungen aufgrund der losen Kopplung größtenteils nur in den isolierten Modulen vorgenommen werden. Der Kern der Anwendung ist in den meisten Fällen schnell stabil und benötigt selten im Laufe der Entwicklung weitere Angleichungen. Geänderte Module können je nach Implementierung auch zur Laufzeit geladen oder hinzugefügt werden, was mögliche Downtime von bereits ausgelieferter Software minimiert [Richards 2015a, 25].

Ein Beispiel dafür stellt die Entwicklung von Betriebssystem-Kernels dar, die auch namensgebend für dieses Architekturmuster

[4]https://github.com/Beleg-6-EAP/demo-modulith-ecommerce

ist. Deren Kern Komponenten sind in der Regel sehr stabil und implementieren vor allem grundlegende Funktionen wie Speicherverwaltung, Prozessverwaltung und I/O-Operationen. Weitere low-level Funktionalitäten wie beispielsweise Geräte Treiber oder Dateisysteme werden als Module in den Kernel geladen und können bei Bedarf hinzugefügt oder entfernt werden, was vor allem für die Unterstützung neuer Hardware wichtig ist.

Auch Entwicklungsumgebungen in der Softwareentwicklung nutzen oft Microkernel Architekturen, um die Support für verschiedene Programmiersprachen und Frameworks zu ermöglichen.

Zwar bietet das Beispiel der E-Commerce Anwendung keinen klassischen Anwendungsfall für Microkernel Architekturen, jedoch kann die Verwendung von Microkernel Architekturen in Teilen der Anwendung trotzdem sinnvoll sein. Sowohl Zahlungs- als auch Versandfunktionalitäten könnten, wie in Abbildung 5 dargestellt, in Module ausgelagert werden, um die Anwendung durch weitere Dienstleister erweitern zu können.

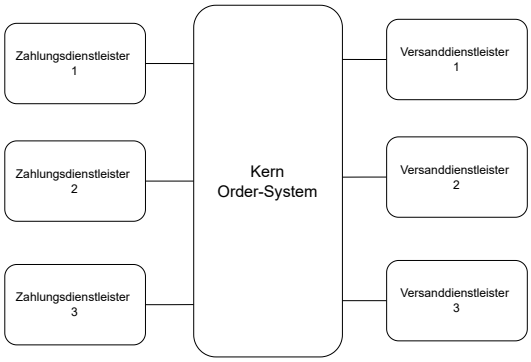

Abb. 5. Aufbau der E-Commerce Anwendung mit Microkernel Architektur

Logik zum Verarbeiten der Zahlungen und Versandinformationen wird dann in den Modulen implementiert, die auf den Kern der Anwendung aufbauen. Dabei ist jedoch ein Großteil der Business-Logik im Kern der Anwendung enthalten, was nicht dem eigentlichen Gedanken der Microkernel Architektur entspricht.

Abgesehen davon erhöhen Microkernel Architekturen inhärent die Testbarkeit der Software, da Module nur lose Kopplung gekoppelt sind. Jedes Modul kann unabhängig voneinander getestet werden und fehlende Module durch Stubs ersetzt werden, wodurch sich während der Entwicklung auf einzelne Module isoliert konzentriert werden kann [Richards 2015a, 26]. Weiterhin können Verhaltensweisen von anderen Modulen durch Mocks simuliert werden, um Testzustände zu erzeugen und das Verhalten der Anwendung zu verifizieren. In Agilen Umgebungen, in denen das Testen von Software eine wichtige Rolle spielt, ist die Verwendung von Microkernel Architekturen daher besonders sinnvoll.

Eine Herausforderung bei der Verwendung von Microkernel Architekturen kann jedoch der Entwurf der Kern-Komponente darstellen. Da alle anderen Module auf den Kern aufbauen, muss dieser sehr sorgfältig und stabil entwickelt werden, um die Funktionalität der gesamten Anwendung zu gewährleisten. Diese Rolle sollten vor allem erfahrene Entwickler übernehmen, da sich Design-Fehler der Kern-Komponente oder Verträge negativ auf die Entwicklung der Module auswirken können. Sollte der Kern der Anwendung angepasst werden, so müssen tendenziell auch alle Module überprüft oder aktualisiert werden, was zu erheblich erhöhten Entwicklungszeiten führen kann und zuvor gewonnene Vorteile der Microkernel Architektur zunichte macht.

Aufgrund der initial hohen Komplexität, die mit der Entwicklung des Kerns einhergeht, stellt die Mikrokernel Architektur nicht die beste Wahl dar, wenn es darum geht schnell eine erste Version der Software auf den Markt zu bekommen. Die wesentlichen Vorteile, die die Microkernel Architektur bietet, zeigen sich erst im späteren Verlauf der Entwicklung, wenn die Anwendung erweitert werden muss. Sowohl Iterationen als auch Auslieferungszeiten sind dann sehr kurz, da Module unabhängig voneinander entwickelt werden und die Anwendung schnell an neue Anforderungen angepasst werden kann. Diese Vorteile können die Architektur in ausgewählten Anwendungsfällen sehr geeignet für agile Entwicklungsumgebungen machen.

3.2 Microservice-Architecture

3.3 Layered-Architecture

Das Problem der redundanten Schnittstellenlogik in Microservices lässt sich durch die Anwendung der Layered Architecture effektiv lösen. Dieses Architekturmuster basiert auf der Grundidee, eine Anwendung in verschiedene Schichten (englisch *layers*) zu gliedern. Dabei bleiben sowohl die Anzahl der Schichten als auch deren spezifischen Aufgaben flexibel und werden nicht durch das Muster vorgeschrieben [Tu 2023, 34].

Diese Flexibilität erlaubt es Unternehmen, die Schichtstruktur individuell an ihre spezifischen Anforderungen anzupassen. Um jedoch Vorteile wie eine verbesserte Wartbarkeit, Skalierbarkeit und Wiederverwendbarkeit zu realisieren, ist die Einhaltung zentraler Prinzipien der Layered Architecture essenziell [Tu 2023, 34].

Ein grundlegendes Prinzip ist die Trennung der Zuständigkeiten (englisch *Separation of Concerns*). Hierbei werden Komponenten mit unterschiedlichen Aufgaben in separate Schichten aufgeteilt, sodass jede Schicht ausschließlich für eine klar definierte und abgeschlossene Funktionalität verantwortlich ist [Tu 2023, 34].

Ein weiteres wesentliches Prinzip ist die Isolation der Schichten (englisch *Layers of Isolation*), welches gewährleistet, dass Änderungen innerhalb einer Schicht lediglich deren eigene Komponenten betreffen und keine Auswirkungen auf andere Schichten haben [Richards 2015b, 3–4].

Die Layered Architecture bietet insbesondere für Unternehmen die Möglichkeit, häufig genutzte Funktionalitäten, wie etwa Authentifizierung oder Logging, in dedizierten Schichten zu kapseln und diese flexibel in verschiedenen Diensten wiederzuverwenden. Dabei sorgt eine klar definierte Schnittstelle jeder Schicht sowohl für eine reibungslose Kommunikation als auch für die Abstraktion der internen Implementierung.

Zusätzlich lässt sich die Layered Architecture nahtlos mit modernen agilen Architekturmustern wie Microservices kombinieren,

wodurch die Agilität eines Unternehmens weiter gesteigert werden kann.

Ein praxisnahes Beispiel hierfür ist das E-Commerce-Beispiel mit Microservices. In der ursprünglichen Implementierung enthielt jeder Service ein eigenes Modul für die Authentifizierung. Diese redundante Struktur führte jedoch zu erheblichen Herausforderungen hinsichtlich Wartbarkeit und Kosten: Änderungen an der Authentifizierungslogik mussten in jedem einzelnen Service separat durchgeführt werden, was den Entwicklungsprozess verlangsamte und unnötig komplizierte.

Eine Lösung für dieses Problem besteht darin, die Authentifizierungslogik in eine zentrale Schicht auszulagern. Diese Schicht kann direkt mit dem API-Gateway kommunizieren und somit eine einheitliche Authentifizierung gewährleisten (siehe Abb. 6).

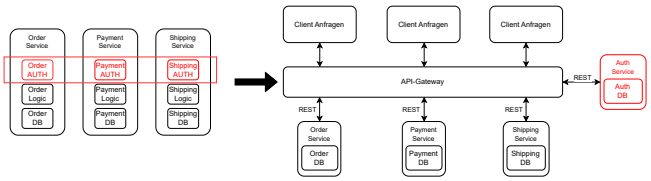

Abb. 6. Layered Microservice Architecture: Beispiel E-Commerce

Mit der Einführung einer zentralen Authentifizierungsschicht kann die Authentifizierungslogik an einer einzigen Stelle gebündelt und bei Bedarf angepasst werden, ohne Änderungen in den einzelnen Services vornehmen zu müssen. Dies bietet insbesondere in Unternehmen mit einer Vielzahl von Services signifikante Zeit- und Kosteneinsparungen.

Ein weiterer Vorteil besteht darin, dass die Authentifizierung für sämtliche Dienste zentral über ein API-Gateway durchgeführt wird, wodurch eine wiederholte Authentifizierung auf Service-Ebene entfällt. Diese Entlastung erlaubt es den einzelnen Services, sich ausschließlich auf ihre Kernfunktionalität zu konzentrieren, während die Authentifizierung ausgelagert ist.

Die vollständige Implementierung des E-Commerce-Beispiels mit geschichteter Microservice-Architektur ist bei GitHub [5] zu finden.

Die klare Trennung der Verantwortlichkeiten fördert die Zusammenarbeit in kleinen, autonomen Teams, da jedes Team unabhängig an einer Schicht arbeiten kann. Zudem ermöglicht das Prinzip der Isolation der Schichten eine höhere Flexibilität gegenüber wechselnden Anforderungen. Durch die Wiederverwendbarkeit in der Layered Architecture wird Mehraufwand vermieden und duplizierter Code reduziert, was zu kürzeren Iterationen und häufigeren Auslieferungen führt.

Das Beispiel verdeutlicht zudem, dass die Layered Architecture sich hervorragend mit agilen Architekturen wie Microservices kombinieren lässt. Diese Synergie trägt entscheidend dazu bei, die Agilität des Systems nachhaltig zu steigern.

3.4 Event-Driven Architecture

Die Event-Driven Architecture wählt als Basis einen anderen Ausgangspunkt als die bisherigen Architekturmuster. Während bei letzteren Komponenten Dienste bereitstellen, welche von anderen Komponenten explizit genutzt werden, verhalten sich Dienstbereitstellende Komponenten in der Event-Driven Architecture reaktiv, werden also implizit von Dienst-konsumierenden Komponenten genutzt [Garlan and Shaw 1994]. Ein System reagiert somit asynchron auf Zustandsänderungen, also Ereignisse in diesem System [Manchana 2021]. Die in dieser Architektur minimalen Einheiten, welche Informationen einer Zustandsänderung kapseln, werden *Events* genannt. Die Idee der impliziten Behandlung von Ereignissen ist nicht neu und taucht erstmals 1994 im von Garlan und Shaw publizierten Papier *„An introduction to Software Architecture“* auf.

Betrachten wir im Folgenden die Basis-Bestandteile der Event-Driven Architecture:

- Ereignis (englisch *Event*): Kapselt Information einer Zustandsänderung eines Systems
- Produzent (englisch *Producer*): Komponente, die Event erzeugt
- Herausgeber (englisch *Publisher*): Komponente, die, von Produzenten erzeugte, Events publiziert
- Konsument (englisch *Consumer*): Komponente, die auf publizierte Events reagiert
- Vermittler (englisch *Mediator*): Komponente zwischen Produzenten und Konsumenten - filtert Events und verteilt diese auf Konsumenten
- Event-Bus: Oft auch *Event-Broker* genannt - bietet die Infrastruktur für die Gesamtheit der Vermittler

Abstrakt kann ein Event als Vertrag zwischen Produzenten und Konsumenten am Event-Bus betrachtet werden. Der Konsument nutzt die Spezifikation des Events am Bus, der Produzent implementiert jene Spezifikation. Abbildung 7 stellt diesen Vertrag dar.

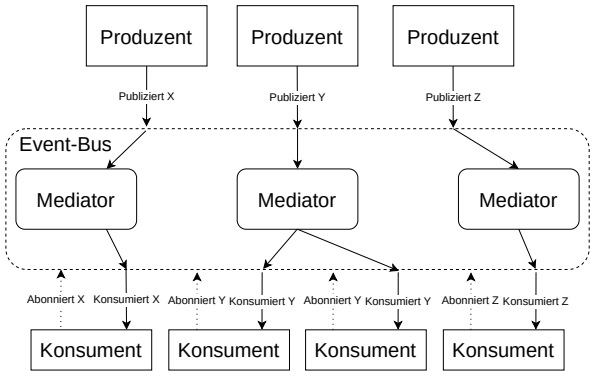

Abb. 7. Vertrag zwischen Produzenten und Konsumenten am Event-Bus

Durch den Vertrag weisen die Events am Event-Bus starke Kohäsion und somit lose Kopplung auf. Diese lose Kopplung minimiert nicht nur kaskadierende Fehler, sondern ermöglicht kleinen und autonomen Entwickler-Teams die klare Abgrenzung von Features und folglich einfach definierbare Iterationen - eine Menge von Events, deren Erzeugung und Konsumierung.

Weiter sind Events oft nah an dem, was Ereignisse in realen Prozessen sind, also domain-driven. Gebündelt ermöglichen obige

[5] https://github.com/Beleg-6-EAP/demo-microservice-ecommerce

Punkte die kontinuierliche Auslieferung von Software in kurzen Intervallen.

Außerdem garantiert die asynchrone Behandlung von Ereignissen zusammen mit der loosen Kopplung hohe Skalierung und die Möglichkeit für Echtzeit-Software. Daher sind Event-Driven Architekturen besonders für datenintensive Echtzeit-Anwendungen wie IoT (Internet of Things) und Analytics geeignet [Siddiqui et al. 2023].

Betrachten wir erneut das E-Commerce-Beispiel aus der Einleitung. Dafür definieren wir drei Arten von Events:

- `OrderCreated`: Ein Event, das genau dann erzeugt wird, wenn eine neue Bestellung aufgegeben wird
- `PaymentProcessed`: Ein Event, das genau dann erzeugt wird, wenn der Bezahlvorgang abgeschlossen wird
- `ShipmentInitiated`: Ein Event, das genau dann erzeugt wird, wenn die Bestellung versandt wird

Analog zur Microservice-Architektur teilen wir die Funktionalitäten in drei verschiedene Dienste auf: `OrderService`, `PaymentService` und `ShipmentService`.

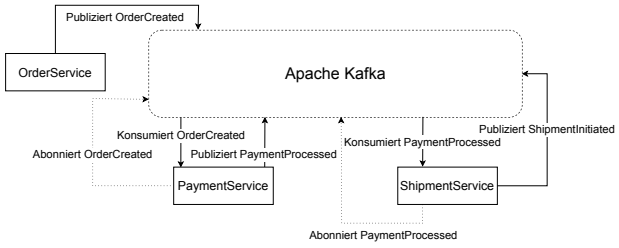

Abb. 8. E-Commerce-Beispiel mit Event-Driven Architecture

Wie Abbildung 8 zeigt, sind alle drei Dienste Produzenten und Publisher, erzeugen also Events und veröffentlichen diese. Die Dienste `PaymentService` und `ShipmentService` sind zudem Konsumenten, sodass ersterer auf Events des Typs `OrderCreated` und zweiterer auf Events des Typs `ShipmentInitiated` reagiert. Eine beispielhafte Implementierung des `PaymentService` mit Apache Kafka als Event-Broker ist im Anhang A.3 zu finden. Die vollständige Implementierung des E-Commerce-Beispiels ist bei GitHub [6] zu finden.

Das Beispiel zeigt, dass die Event-Driven Architektur mit weiteren agilen Strukturen wie Microservices kombiniert werden kann, was die Agilität der Architektur weiter erhöht. Die damit einhergehende Komplexität stellt teilweise hohe Anforderungen an die Entwickler. Aufgrund der Asynchronität der Behandlung von Ereignissen ist die Testung des Systems meist schwer und die Fehlerbehandlung essentiell. Mögliche Problemquellen schließen dabei unter anderem Event-Verlust, erhöhte Latenz und Inkonsistenz ein. Die hohen Anforderungen an die Entwickler verlangen viel Vertrauen in jene, einer der zentralen Punkte des agilen Manifests [Michl 2018]. Insgesamt weist die Event-Driven Architecture also eine sehr hohe Agilität auf und ist damit besonders für moderne Software und ihre stetig wechselnden Anforderungen geeignet.

[6] https://github.com/Beleg-6-EAP/demo-eda-ecommerce

3.5 Cloud-Native Architecture

Die Cloud-Native Architecture beruht auf der Annahme, dass die Infrastruktur in ständigem Wandel ist und die Auslagerung jener in die Cloud mehr Agilität schafft [Gannon et al. 2017]. Als Cloud wird in diesem Fall die Infrastruktur eines oder mehrerer Cloud-Vendors bezeichnet. Ein Cloud-Vendor bietet seine Infrastruktur und deren Verwaltung dabei gegen eine Gebühr an. Unter anderem offeriert er:

- Die globale Nutzung von Ressourcen durch Geo-Redundanz und somit starke Verteilung sowie hohe Verfügbarkeit von Software,
- dynamische Skalierung bereitgestellter Ressourcen basierend auf der Nachfrage von Software (Auto-Scaling),
- nutzungsbedingte Gebühren, sodass nur tatsächlich verwendete Ressourcen bezahlt werden (Pay-as-you-go),
- unterbrechungsfreie Updates von Software (Zero Downtime).

Als cloud-nativ wird hierbei all jene Software bezeichnet, welche explizit für die Cloud entwickelt wurde. Grundsätzlich baut die cloud-native Architektur auf die in diesem Artikel bereits in Abschnitt 3.2 erklärte Microservice-Architektur und die in Abschnitt 3.4 erklärte Event-Driven Architecture auf. Zusätzlich kommen sogenannte *Fully Managed Cloud-Services* hinzu, was cloud-spezifische und vom Cloud-Vendor vollständig verwaltete Dienste sind. Jene umfassen alle von Datenbanken über Message-Queues bis hin zu Serverless-Functions und viele mehr. Charakterisiert werden diese durch die Eigenschaft, dass sich der Entwickler gänzlich auf die Business-Logik konzentrieren kann, da der Cloud-Provider die vollständige Verwaltung der Infrastruktur übernimmt. Zentral dabei ist der Aspekt der Containerisierung, bei welchem jede Komponente eines Systems in einen Container gepackt wird. Der Cloud-Vendor kann die Gesamtheit der Container dann dynamisch orchestrieren und somit gezielt den Ressourcenverbrauch optimieren.

Im Folgenden greifen wir erneut das Beispiel E-Commerce aus der Einleitung auf und betrachten die Anpassung der in Abschnitt 3.4 angewendeten Event-Driven-Architecture auf die Cloud-Native Architecture mit Cloud-Vendor AWS.

Dabei ersetzen wir den dort verwendeten Broker durch eine *EventBridge* [7] - ein serverless Cloud-Service von AWS zum Routen von Ereignissen. Weiter werden die drei Services für Aufgeben einer Bestellung, Bearbeitung der Bezahlung und Initiierung des Versands als *Lambda* [8], also Serverless-Function implementiert. Abbildung 9 stellt diese cloud-native Architektur dar. Eine beispielhafte Implementierung des `ProcessPaymentLambda`s ist im Anhang A.4 und auf GitHub[9] zu finden.

Damit kombiniert die Cloud-Native Architecture die agilen Vorteile der Microservice- und Event-Driven Architecture. Weiter ermöglichen die Cloud-Vendors nahtlose und einfache Möglichkeiten zur Auslieferung der Software, wie beispielsweise Code-Deploy [10] bei AWS. Die Option, allen Fokus von der Infrastruktur auf die Entwicklung zu legen, ermöglicht gemeinsam mit der einfachen Auslieferung kurze Iterationen in agilen Teams und somit noch

[7] https://aws.amazon.com/de/eventbridge/, abgerufen am 06.01.2025
[8] https://aws.amazon.com/de/lambda/, abgerufen am 06.01.2025
[9] https://github.com/Beleg-6-EAP/demo-cloud-native-ecommerce
[10] https://aws.amazon.com/de/codedeploy/, abgerufen am 06.01.2024

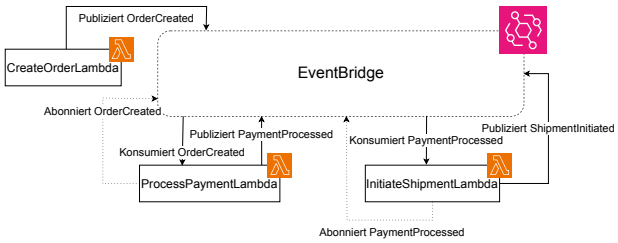

Abb. 9. E-Commerce-Beispiel mit Cloud-Native Architecture

kürzere Zyklen in der Bereitstellung von Software. Dabei ist besonders die Time-to-Market sehr gering, sodass Tech-Start-Ups häufig cloud-native Architekturen für ihre Software wählen.

Nicht zu vergessen ist zudem die extrem hohe Flexibilität der eingesetzten Ressourcen durch das Auto-Scaling. Das damit verbundene Pay-as-you-go ermöglicht weiter auch finanzielle Agilität, wodurch zum Beispiel keine Vorab-Investitionen für Infrastruktur notwendig sind. Jedoch ist zu betonen, dass aufgrund dessen auch Kostenrisiken durch möglicherweise unerwartete, extrem hohe Nachfrage der Software bestehen. Ebenfalls nicht zu vernachlässigen ist die enge Bindung an den Cloud-Vendor durch Verwendung seiner spezifischen Cloud-Dienste. Im Fall einer Migration zu einem anderen Cloud-Vendor könnten sowohl hohe Kosten beim vorherigen Cloud-Vendor, als auch Entwicklungskosten durch die notwendige Anpassung der genutzten spezifischen Cloud-Dienste anfallen. Obwohl das die Wahl des Cloud-Anbieters und der damit verbunden Entwicklung weniger agil macht, gilt die Cloud-Native Architecture besonders aufgrund der Auslagerung der Infrastruktur als eine - und vermutlich die agilste Architektur für moderne Software.

4 FALLSTUDIEN UND PRAXISBEISPIELE

Blah …

5 DISKUSSION

6 ZUSAMMENFASSUNG UND AUSBLICK

LITERATUR

Kalpesh Barde. 2023. Modular Monoliths: Revolutionizing Software Architecture for Efficient Payment Systems in Fintech. (2023). https://doi.org/10.1145/3643657.3643911

Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. 2022. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. *IEEE Access* 10 (2022), 20357–20374. https://doi.org/10.1109/ACCESS.2022.3152803

Rui Chen, Shanshan Li, and Zheng Li. 2017. From Monolith to Microservices: A Dataflow-Driven Approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. 466–475. https://doi.org/10.1109/APSEC.2017.53

Dennis Gannon, Roger Barga, and Neel Sundaresan. 2017. Cloud-Native Applications. *IEEE Cloud Computing* 4, 5 (2017), 16–21. https://doi.org/10.1109/MCC.2017.4250939

David Garlan and Mary Shaw. 1994. *An Introduction to Software Architecture.* Technical Report CMU/SEI-94-TR-021. https://insights.sei.cmu.edu/library/an-introduction-to-software-architecture/ Accessed: 2025-Jan-2.

Danny Greefhorst and Erik Proper. 2011. *The Role of Enterprise Architecture.* Springer Berlin Heidelberg, Berlin, Heidelberg, 7–29. https://doi.org/10.1007/978-3-642-20279-7_2

Ramakrishna Manchana. 2021. Event-Driven Architecture: Building Responsive and Scalable Systems for Modern Industries. *International Journal of Science and Research (IJSR)* 10 (01 2021), 1706–1716. https://doi.org/10.21275/SR24820051042

Thomas Michl. 2018. *Das agile Manifest – eine Einführung.* Springer Berlin Heidelberg, Berlin, Heidelberg, 3–13. https://doi.org/10.1007/978-3-662-57699-1_1

Mark Richards. 2015a. *Software Architecture Patterns.* O'Reilly Media, Inc. value pages.

Mark Richards. 2015b. *Software architecture patterns.* Vol. 4. O'Reilly Media, Incorporated 1005 Gravenstein Highway North, Sebastopol, CA ….

Tasneem Salah, M. Jamal Zemerly, Chan Yeob Yeun, Mahmoud Al-Qutayri, and Yousof Al-Hammadi. 2016. The evolution of distributed systems towards microservices architecture. In *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*. 318–325. https://doi.org/10.1109/ICITST.2016.7856721

Nada Salaheddin and Nuredin Ahmed. 2022. MICROSERVICES VS. MONOLITHIC ARCHITECTURE [THE DIFFERENTIAL STRUCTURE BETWEEN TWO ARCHITECTURES]. *MINAR International Journal of Applied Sciences and Technology* 4 (10 2022), 484–490. https://doi.org/10.47832/2717-8234.12.47

Hassaan Siddiqui, Ferhat Khendek, and Maria Toeroe. 2023. Microservices based architectures for IoT systems - State-of-the-art review. *Internet of Things* 23 (2023), 100854. https://doi.org/10.1016/j.iot.2023.100854

Ruoyu Su and Xiaozhou Li. 2024. Modular Monolith: Is This the Trend in Software Architecture?. In *Proceedings of the 1st International Workshop on New Trends in Software Architecture* (Lisbon, Portugal) *(SATrends '24)*. Association for Computing Machinery, New York, NY, USA, 10–13. https://doi.org/10.1145/3643657.3643911

Zhenan Tu. 2023. Research on the Application of Layered Architecture in Computer Software Development. *Journal of Computing and Electronic Information Management* 11 (11 2023), 34–38. https://doi.org/ojs/index.php/jceim/article/view/14398

A CODE-BEISPIELE

A.1 Monolithic Architecture

```java
package demo.monolith.service;

import demo.monolith.model.Payment;
import demo.monolith.repository.PaymentRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.UUID;

@Service
@RequiredArgsConstructor
public class PaymentService {

    private final PaymentRepository paymentRepository;

    public void processPayment(String orderId) {
        final Payment payment = new Payment(UUID.randomUUID().toString(), orderId, true);
        paymentRepository.save(payment);
    }

    public List<Payment> getAll() {
        return paymentRepository.findAll();
    }
}
```

Listing 1. Service-Implementierung des `PaymentService` in Java Spring Boot 3.4.1

A.2 Modular Monolithic Architecture

```java
package demo.modulith.order.internal;

import demo.modulith.order.OrderService;
import demo.modulith.payment.PaymentService;
import demo.modulith.payment.internal.PaymentServiceImpl;
import demo.modulith.shipment.ShipmentService;
import demo.modulith.shipment.internal.ShipmentServiceImpl;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
@RequiredArgsConstructor
public class OrderServiceImpl implements OrderService {

    private final OrderRepository orderRepository;
    private final PaymentService paymentService;
    private final ShipmentService shipmentService;

    public Order create(Order order) {
        final Order savedOrder = orderRepository.save(order);
        paymentService.processPayment(savedOrder.getId());
        shipmentService.handleShipping(savedOrder.getId());

        return savedOrder;
    }

    public List<Order> getAll() {
        return orderRepository.findAll();
    }
}
```

Listing 2. Service-Implementierung des `OrderService` in Java Spring Boot 3.4.1

A.3 Event-Driven Architecture

```java
package demo.eda.service;

import demo.eda.event.OrderCreatedEvent;
import demo.eda.event.PaymentProcessedEvent;
import demo.eda.model.Payment;
import demo.eda.repository.PaymentRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.core.reactive.ReactiveKafkaProducerTemplate;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;

import java.util.UUID;

@Service
@RequiredArgsConstructor
public class PaymentService {

    private final PaymentRepository paymentRepository;
    private final ReactiveKafkaProducerTemplate<String, PaymentProcessedEvent> producer;

    @KafkaListener(topics = OrderCreatedEvent.TOPIC, groupId = "payment-service")
    public Mono<Void> processPayment(OrderCreatedEvent event) {
        final Payment payment = new Payment(UUID.randomUUID().toString(), event.getOrderId(), true);
        return paymentRepository.save(payment).flatMap(savedPayment -> {
            final PaymentProcessedEvent paymentEvent =
                    new PaymentProcessedEvent(
                            savedPayment.getOrderId(),
                            savedPayment.getId(),
                            savedPayment.isSuccess());
            return producer.send(PaymentProcessedEvent.TOPIC, paymentEvent).then();
        }).then();
    }
}
```

Listing 3. Service-Implementierung des `PaymentService` in Java Spring Boot 3.4.1 mit Apache Kafka als Event-Broker

A.4 Cloud-native Architecture

```haskell
{-# LANGUAGE GHC2024 #-}
{-# LANGUAGE DeriveAnyClass #-}

module PaymentService where

import AWS.Lambda.Runtime (ioRuntime)
import Data.Aeson
import Data.UUID.V4 (nextRandom)
import Data.UUID (toString)
import GHC.Generics (Generic)

data OrderCreatedEvent = OrderCreatedEvent
  { orderId :: String
  , userId  :: String
  , amount  :: Double
  } deriving stock (Show, Generic)
  deriving anyclass (FromJSON, ToJSON)

data Payment = Payment
  { id             :: String
  , paymentOrderId :: String
  , success        :: Bool
  } deriving stock (Show, Generic)
  deriving anyclass (FromJSON, ToJSON)

orderCreatedEventHandler :: OrderCreatedEvent -> IO ()
orderCreatedEventHandler event = do
  uId <- nextRandom
  let payment = Payment { id = toString uId, paymentOrderId = orderId event, success = True }
  -- imagine here that payment always succeeds and we serialize it successfully
  -- imagine further serialization into e.g. Amazon RDS which would trigger a PaymentProcessedEvent
  return ()

-- entry point for aws lambda
main :: IO ()
main = ioRuntime $ fmap Right . orderCreatedEventHandler
```

Listing 4. Implementierung des `ProcessPaymentLambda`s in Haskell

B ÜBUNGSAUFGABEN

B.1 Übungsaufgabe 1

Das E-Commerce-Beispiel aus der Einleitung soll um Nutzer-Authentifizierung erweitert werden. Sie haben sich zuvor für eine Microservice-Architektur entschieden und die in der Einleitung genannten Anforderungen implementiert. Die Authentifizierung wird in verschiedenen Komponenten benötigt. Erläutern Sie, wie Sie die Authentifizierung in die Architektur integrieren.

B.2 Übungsaufgabe 2

Das E-Commerce-Beispiel aus der Einleitung soll um Logging erweitert werden. Sie haben sich zuvor für eine cloud-native Event-Driven-Architektur entschieden und die in der Einleitung genannten Anforderungen implementiert. Das Logging wird in verschiedenen Komponenten benötigt. Erläutern Sie, wie Sie das Logging in die Architektur integrieren. Bedenken Sie dabei, dass Logs aus verschiedenen Komponenten möglicherweise zur Auswertung zusammengeführt werden müssen und somit die Reihenfolge von Logs relevant ist.

B.3 Übungsaufgabe 3

Das E-Commerce-Beispiel aus der Einleitung soll um ein E-Mail-Notifikationssystem erweitert werden. Dieses soll Nutzern E-Mails bei jeder Statusänderung einer ihrer Bestellungen zustellen. Untersuchen Sie für alle in diesem Papier betrachteten Architekturen, wie das E-Mail-Notifikationssystem in die bestehende Architektur integriert werden kann und welche Vor- und Nachteile diese Integration in die jeweilige Architektur mit sich bringt. Für welche der Architekturen ist die Integration des E-Mail-Notifikationssystems am einfachsten?